Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 127 (2005) 5–16

www.elsevier.com/locate/entcs

Leveraging UML Profiles to Generate Plugins From Visual Model Transformations

Hans Schippers*, Pieter Van Gorp, Dirk Janssens

Formal Techniques in Software Engineering
Universiteit Antwerpen, Belgium
{hans.schippers,pieter.vangorp,dirk.janssens}@ua.ac.be
* *Research Assistant of the Research Foundation - Flanders (FWO - Vlaanderen)*

Abstract

Model transformation is a fundamental technology in the MDA. Therefore, model transformations should be treated as first class entities, that is, models. One could use the metamodel of SDM, a graph based object transformation language, as the metamodel of such transformation models. However, there are two problems associated with this. First, SDM has a non-standardized metamodel, meaning a specific tool (Fujaba) would be needed to write transformation specifications. Secondly, due to assumptions of the code generator, the transformations could only be deployed on the Fujaba tool itself. In this paper, we describe how these issues have been overcome through the development of a template based code generator that translates instances of a UML profile for SDM to complete model transformation code that complies to the JMI standard. We have validated this approach by specifying a simple visual refactoring in one UML tool and deploying the generated plugin on another UML tool.

Keywords: Refactoring, Model Transformation, SDM, JMI

1 Introduction

As Sendall and Kozaczynski state [16], model transformation can be seen as the *heart and soul* of model driven software development. In terms of OMG's Model Driven Architecture (MDA [13]), PIM-to-PSM transformations come to mind immediately, but that is only half the story. Indeed, beside these *refinements* (a special kind of *translations*), there is another important class of model transformations: *rephrasings* [6]. These are transformations within the same metamodel (intra-metamodel), which could be applied to change a model because of evolving requirements, or to enhance a model's internal structure

doi:10.1016/j.entcs.2004.08.029

without modifying its external behavior (refactoring). Recent experiments [5] have shown that Fujaba's Story Driven Modeling (SDM [4]) can be used as a language for developing transformations of this class.

However, one was restricted to Fujaba as its development environment for two reasons. The first has to do with the fact that SDM is an independent, non-standard metamodel, and is only implicitly present in the Fujaba source code. Therefore, SDM specifications could only be written with the Fujaba editor. The second problem is that model transformations developed with Fujaba can only be deployed on the Fujaba repository itself. Its cause is that the Fujaba code generator only integrates with code complying to a Fujaba proprietary API. More specifically, it generates code that is based on a specific association framework. Obviously, these issues stand in the way of the approach becoming mainstream. They have been overcome by, on the one hand, designing a UML profile for SDM, implying that any CASE tool can be used for the development of transformation models, and on the other hand, developing a new code generator for the resulting metamodel [15]. The latter was handled in such a way that the part which depends on the target platform can easily be replaced.

This paper describes this work, and is organized as follows: First, we provide the required background information by summarizing the related MDA standards. Next, the architecture of the code generator is described, which is then illustrated by an example of a model transformation. Finally, conclusions are drawn and potential future work is discussed.

2 MDA Standards

As explained above, in spite of Fujaba's value in validating numerous model management techniques [12,11,17], the tool lacks standardization. More precisely, its code generator reads its input in a proprietary way, from a non-standard repository, and generates output code for the same repository, again making use of its non-standard API. In what follows, some standards and concepts, which have been used to solve this problem, are presented.

2.1 Meta Object Facility (MOF)

The MOF standard [14] essentially defines a four-layered *metadata architecture*, as shown in Fig. 1. At the top (M3) is the meta-metamodel (also known as the MOF model), a universal language to define metamodels (M2). Metamodels are themselves languages used to define models (M1), which in turn describe the actual data (M0). In other words, the model at level M_n provides a description of some common characteristics of the data at level M_{n-1}. One

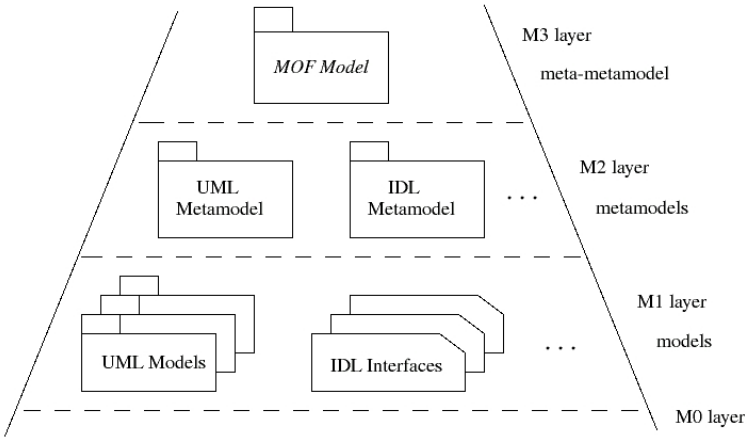


Fig. 1. MOF Metadata Architecture

specific set of data, conforming to a model, is called an *instance* of that model. As shown in Fig. 2, "model" and "metamodel" are relative concepts: a metamodel can easily be seen as a model of a model, while the meta-metamodel can be seen as the model of a metamodel. The Unified Modeling Language (UML) for example, can be formalized by a metamodel (an instance of the MOF model). UML can be used to specify class diagrams, activity diagrams, etc. which, as a consequence, can be parsed to instances of the UML metamodel, or models at layer M1.

The MOF model is designed to be universal: it should be adequate to describe *any* metamodel, including its own metamodel (which is the MOF model itself). Since "metamodel" is a relative concept, the meta-metamodel (i.e., the MOF model) can be seen as the metamodel of all metamodels on the M2 layer. In this sense, the MOF model can be stored on the M2 level. However, one can only reason about all metamodels in a standard way by agreeing on one model for meta-metamodeling. Therefore, the MOF model is logically considered to be on the M3 level. In the context of this paper, the most important merit of the MOF standard is that it allows the creation of tools for model analysis and manipulation, which only depend on the MOF model, but not on any specific metamodel. In particular, a MOF repository can be developed, which supports the storage of any MOF-compliant models and metamodels. The open source NetBeans Metadata Repository (MDR [10]) does just that, and was therefore a logical candidate for the new code generator.

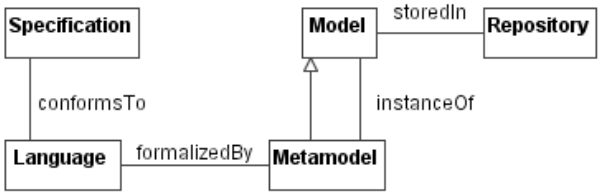


Fig. 2. The relationship between languages, models, metamodels and repositories.

2.2 Java Metadata Interface (JMI)

The MOF standard on itself is not the whole story, since it does not define how models can be accessed from source code. Or rather, it does, but only for CORBA IDL, and not for any other language. As its name suggests, the Java Metadata Interface (JMI) standard [9] provides a solution here, by actually *mapping* MOF to Java. More specifically, JMI defines one or more Java entities for each MOF construct, thus introducing a standard API for model access. For example, a MOF class is mapped to two Java interfaces: one "factory" (or "class proxy") interface for constructing objects and one "instance" interface for manipulating them. By applying this mapping to a metamodel, which of course consists of these MOF constructs (as it is a MOF instance), a metamodel-specific set of interfaces is obtained, through which any instance of this metamodel can be accessed and manipulated. In case of UML, for example, these interfaces can be used to add a new UML class to a model of a class diagram, or find an existing UML association and delete it. In addition, there is also a unique set of reflective interfaces, which offers the same possibilities, but without having to use metamodel-specific code.

In order to understand that a standard like JMI is sufficient to build model-manipulating tools in a metamodel- (and model-) independent way, the following two points are crucial:

(i) *model* manipulation must always be carried out through *metamodel* interfaces. For example, a UML class diagram can only be seen in terms of UML classes, UML attributes, UML associations, etc. which are all concepts from the UML metamodel, and as such are present in the UML-specific interfaces.

(ii) indirectly making use of metamodel-specific interfaces, does not make a tool metamodel-dependent. In the following section, it will be shown that the code generator can produce metamodel-specific code by relying on the JMI *mapping rules* only. Obviously, it is desirable that the generated code *is* metamodel-specific, as each model transformation is metamodel-specific as well.

3 Architecture

This section describes the overall architecture of JCMTG, that is, the JMI Compliant Model Transformer Generator, a standards-based alternative for Fujaba's proprietary way of handling model transformations.

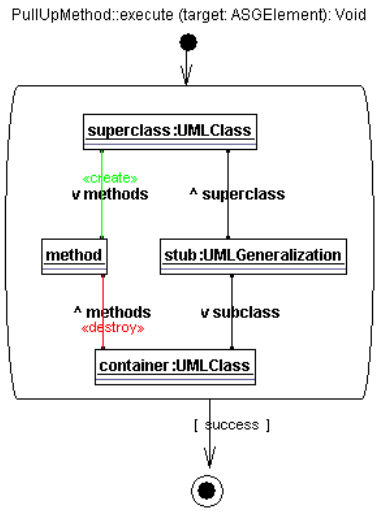


Fig. 3. Example SDM specification edited with the Fujaba UML tool.

3.1 UML Profile for SDM

As already indicated in Section 1, the trouble with SDM (as it is used in Fujaba) is that its syntax as well as its semantics are non-standard, even though they both resemble their UML counterpart. The latter is illustrated in Fig. 3, which displays an excerpt of an SDM specification in Fujaba. While elements of both activity diagrams and collaboration diagrams can easily be recognized, it is clear that no CASE tool is capable of drawing similar diagrams, as UML does not support nesting in that way. Furthermore, the storage of SDM instances in Fujaba is also non-standard. Both aspects of this problem were tackled in JCMTG by designing a UML profile for SDM. In practice, this comes down to *mapping* each SDM construct to a UML alternative. Additionally, stereotypes have been used to differentiate between several variants of the same basic SDM constructs (for example *forEach* activities versus *code* activities versus normal *story* activities). For the control flow part, this proved to be quite straightforward, because of the support of activity diagrams in UML. For the so-called *transformation primitives*, which

actually resemble collaboration diagrams, UML class diagrams have been chosen instead, as these often seem to offer more visual features, such as attribute assignments. An excerpt of the SDM-to-UML mapping is given in Table 1. It should now be clear that the UML profile allows that, on the one hand, SDM specifications can be drawn in any CASE tool, while on the other hand, since UML is a MOF instance, a standards-based MOF repository (in particular, MDR) can be employed for storage purposes.

SDM Construct	**UML Construct**
Story Activity	ActionState
ForEach Activity	ActionState with «for each» stereotype
Unbound object	UmlClass
Bound Object	UmlClass with «bound» stereotype

Table 1
Extract from SDM-to-UML mapping

3.2 Generation of Transformation Code

An overview of the actual code generation process is displayed in Fig. 4. The MOF repository (MDR) plays an important part, and could be seen as the starting point, as it holds the transformation specification (or transformation model). Since MDR provides a JMI API, this specification can be analyzed in a standardized way by the code generator engine. The open source AndroMDA [1] code generator was chosen for this task, at the heart of which is in fact a set of dynamic content templates. These provide a "skeleton" of the generated code, which is filled in depending on the information in the transformation model.

Fig. 4 illustrates that this transformation model is defined on the metamodel of the models to be transformed: since we are implementing translations, the in- and output metamodel of the transformations is the same. The artifact resulting from template instantiation is a Java source file, containing metamodel-specific JMI code. This code can analyze and transform any model instantiating this metamodel. In practice, a transformation writer defines model checks and transformations as path navigations and rewritings over a graph structure of this metamodel. One can define a type graph as a class diagram either manually or reverse engineer it from the target repository sources. Ideally, class diagrams of mainstream metamodels (e.g., UML 1.5, 2.0, ...) would be shared by the transformation community.

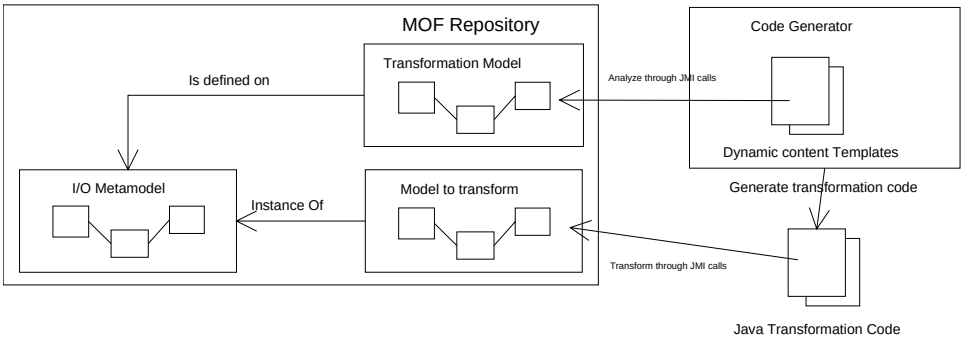


Fig. 4. JCMTG Architecture

Note that, even though JMI sets a standard, it may be useful to generate code for other platforms (after all this is MDA). In that case, the dynamic content templates can easily be replaced by a different set, which target a new platform like repositories conforming to the Eclipse Modeling Framework (EMF [8]).

3.3 Constraints at Two Levels

There are two levels in the transformation process where constraints should be checked.

In order to guarantee generation of correct code, it is important that a transformation model can be checked for well-formedness. Indeed, UML on itself has quite loose semantics, and the interpretation specific to SDM is obviously not captured at all. Ideally, a MOF repository should allow direct verification of such metamodel well-formedness rules (OCL would make a good candidate here, since it is a MOF instance itself), but unfortunately, this is currently not implemented in the MDR. Therefore, the well-formedness of transformation models can currently only be verified after they have been serialized to XMI and imported in a dedicated OCL constraint checker like OCLE [2].

Yet, there is another level where constraints come into play, namely within a transformation specification. Indeed, it may be desirable to only execute (part of) the transformation if a certain complex condition is satisfied, or perform different actions depending on the truth value of such a condition. Thus, OCL is relevant in that context too. However, once more, tool support is lacking. Pragmatically, JCMTG adopted Java conditions instead. Note that this is not ideal, as it makes the transformation model depend upon the target platform (JMI, EMF, ...), which prevents large scale reuse of transformation specifications. The Dresden OCL Toolkit [7] seems to be a promising alter-

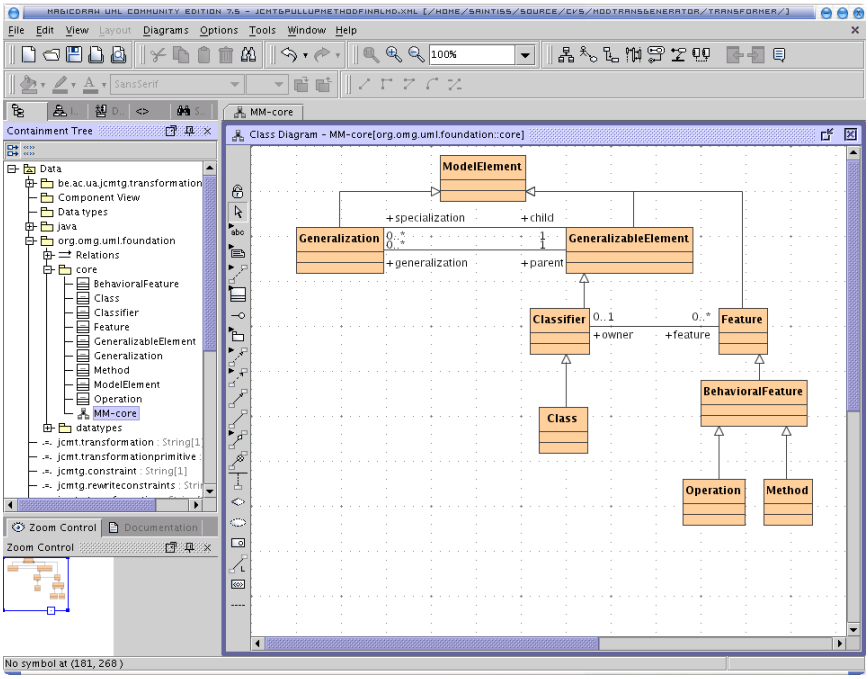


Fig. 5. Fragment of the UML 1.5 metamodel.

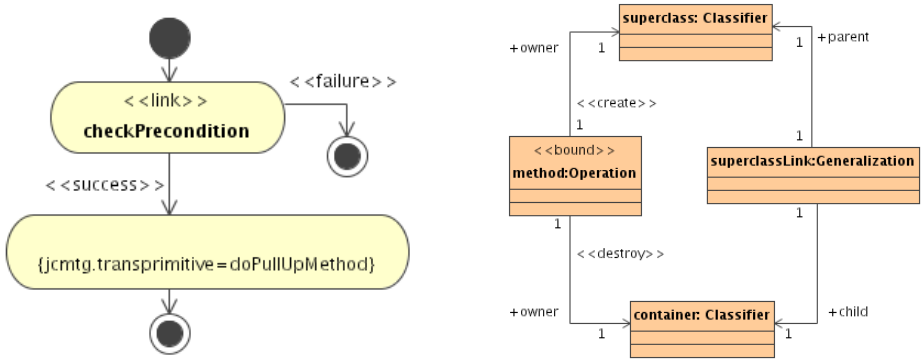
native, as it should be capable of parsing OCL constraints, and evaluating them directly on a MOF repository. Unfortunately, the parser is still under development.

4 Example: Pull Up Method

Demonstrating how everything fits together is perhaps best done by means of an example. Consider the so-called "pull up method" refactoring, which basically just moves a method of class A to class B, where A inherits from B. The transformation is defined on a fragment of the UML 1.5 metamodel shown in Fig. 5.

The corresponding transformation model is illustrated in Fig. 6 where, just as in the Fujaba example (Fig. 3 on page 5), two main parts can be distinguished, albeit not nested anymore. Note that in the UML profile, the reference between the two parts is maintained by means of a tagged value.

The transformation flow is quite straightforward. First a precondition, which basically just makes sure it makes sense to apply the refactoring, is checked, and only if it returned "true", the actual transformation is carried out. The latter, the so-called "transformation primitive" specifies a graph



(a) (b)

Fig. 6. PUM transformation model

rewriting that is displayed in Fig. 6 (b). It illustrates the main idea behind SDM: initially, a set of objects matching the structure given in the primitive, is searched for. More precisely, a method should be found, which belongs to a certain class "container" (this can be checked via the "owner" association in the UML metamodel). Additionally, "container" should have a superclass "superclass", which can be reached by navigating through the UML metamodel some more. If, and only if, such a structure can be matched, the "owner" link to "container" is removed, and an "owner" link to "superclass" is established, signaling successful completion of the transformation.

After specifying this transformation in a UML 1.5 compliant tool, one exports it to XMI. JCMTG then generates a complete plugin for the Poseidon tool, which has a JMI compliant UML 1.5 repository. Fig. 7 displays the plugin popup appearing when one right-clicks on a method. As specified in the abstract transformation, the method will only be pulled up if the precondition of the refactoring is met.

5 Conclusion and Future Work

It should be clear that the elaborated approach solves the two significant issues from which Fujaba suffers. First, the UML profile ensures the possible usage of any UML 1.5 compliant CASE tool to draw transformation models, as well as standardized model access and storage. Second, the employment of pluggable dynamic content templates guarantees independence from any specific target platform. Nevertheless, JCMTG is only very young, and many aspects would

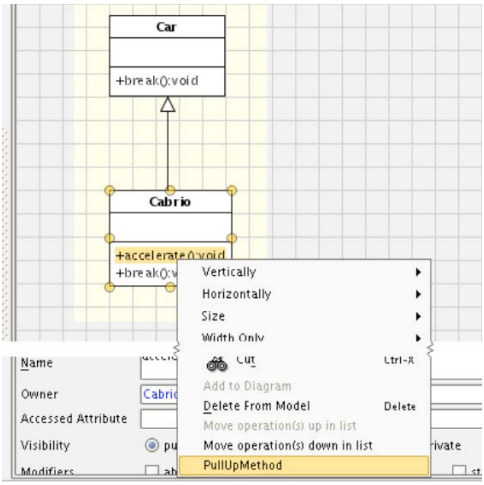


Fig. 7. Screenshot from the UML CASE tool plugin generated from the abstract transformation specification.

benefit from certain improvements. As already mentioned in Section 3.3 for example, better OCL tool support both at metamodel- and model-level, would enable integration of constraint checking. Additionally, expressiveness of SDM could be questioned, especially in the context of inter-metamodel transformations, that is, transforming instances from one metamodel to become instances of another metamodel. In this light, but also when considering very complex transformations, it might be desirable to add additional constructs to the language. Another important issue has to do with the dynamic content templates. Although they can easily be replaced, chances are that a significant amount of any other set of templates would be very similar, if not identical. Therefore, it would probably be worthwhile to investigate how extra levels of abstraction can be introduced between the transformation model and the transformation code. In this light, the transformation engine project at INRIA [3] is very promising, as it introduces a so-called "pivot-metamodel". This is a rather low-level metamodel, from which code generation is straightforward. The idea is that a transformation model is first translated to this pivot, instead of generating code immediately. This would ensure that a change of target platform only causes the (trivial) step from pivot to code to be replaced. Finally, a note on the concrete syntax. The argument that the Fujaba notation was more elegant than the, admittedly somewhat artificial, UML notation is probably valid. It is, however, important to distinguish between concrete and abstract syntax. Only the latter is really tied to UML, so nothing prevents a tool developer from creating an environment with Fujaba's concrete syntax, and transform this behind the scenes to fit into the UML profile for stor-

age. That way, the possibility that for instance an abundance of stereotypes would make transformation specifications less readable, would not be an issue anymore.

References

[1] M. Bohlen. AndroMDA - from UML to Deployable Components, version 2.1.2, 2003. <http://andromda.sourceforge.net>.

[2] D. Chiorean et al. Object constraint language environment (OCLE), version 2.02, 2004. <http://lci.cs.ubbcluj.ro/ocle>.

[3] Institut National de Récherche en Informatique et en Automatique (INRIA). MTL Model Transformation Engine, 2004. <http://modelware.inria.fr>.

[4] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)*, volume 1764 of *LNCS*, pages 296–309. Springer Verlag, November 1998.

[5] Pieter Van Gorp, Niels Van Eetvelde, and Dirk Janssens. Implementing Refactorings as Graph Rewrite Rules on a Platform Independent Metamodel. In *Proceedings of the 1st International Fujaba Days*, University of Kassel, Germany, October 2003.

[6] P. Van Gorp. Write Once, Deploy N: a Performance Oriented MDA Case Study. In *Dagstuhl Seminar 04101 - Language Engineering for Model-Driven Software Development*, march 2004.

[7] S. Loecher and S. Ocke. A Metamodel-Based OCL-Compiler for UML and MOF. In *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA*, October 2003.

[8] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks. International Business Machines, January 2004.

[9] Sun Microsystems. Java Metadata Interface Specification, June 2002. document ID JSR-40.

[10] Sun Microsystems. NetBeans Metadata Repository, 2002. <http://mdr.netbeans.org/>.

[11] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the* 24^{th} *International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348. ACM Press, May 2002.

[12] J. Niere and A. Zündorf. Using fujaba for the development of production control systems. In *Proc. of International Workshop and Symposium on Applications Of Graph Transformations With Industrial Relevance (AGTIVE), Kerkrade, The Netherlands*, LNCS 1779. Springer Verlag, 1999.

[13] Object Management Group. Model Driven Architecture (MDA), July 2001. document ID ormsc/01-07-01.

[14] Object Management Group. Meta-Object Facility Specification, April 2002. version 1.4. document ID formal/02-04-03.

[15] Hans Schippers. JMI Conforme Modeltransformator Generator. Master's thesis, University of Antwerp, Belgium, 2004.

[16] S. Sendall and W. Kozaczynski. Model Transformation - The Heart and Soul of Model-Driven Software Development. *IEEE Software, Special Issue on Model Driven Software Development*, pages 42–45, Sept/Oct 2003.

[17] Robert Wagner, Holger Giese, and Ulrich Nickel. A plug-in for flexible and incremental consistency management. In *Proc. of the International Conference on the Unified Modeling Language 2003 (Workshop 7: Consistency Problems in UML-based Software Development), San Francisco, USA*, October 2003.